Bruno's Ramblings
A freethinker's view on programming

Fibers and Threads in node.js – what for?

Posted on March 11, 2012

I like node.js, and I'm not the only one, obviously! I like it primarily for two things: it is *simple* and it is *very fast*. I already said it many times but one more won't hurt.

Before working with node, I had spent many years working with threaded application servers. This was fun sometimes but it was also often frustrating: so many APIs to learn, so much code to write, so many risks with critical sections and deadlocks, such a waste of costly system resources (stacks, locks), etc. Node came as a breath of fresh air: a simple event loop and callbacks. You can do a lot with so little, and it really flies!

But it does not look like we managed to eradicate threads. They keep coming back. At the beginning of last year Marcel Laverdet opened Pandora's box by releasing `node-fibers`: his threads are a little greener than our old ones but they have some similarities with them. And this week the box got wide open as Jorge Chamorro Bieling released `threads_a_gogo`, an implementation of *real* threads for node.js.

Isn't that awful? We were perfectly happy with the event loop and callbacks, and now we have to deal with threads and all their complexities again. Why on earth? Can't we stop the thread cancer before it kills us!

Well. First, things aren't so bad because fibers and threads did not make it into node's core. The core is still relying only on the event loop and callbacks. And it is probably better this way.

And then maybe we need to overcome our natural aversion for threads and their complexities. Maybe these new threads aren't so complex after all. And maybe they solve real problems. This is what I'm going to explore in this post.

Threads and Fibers

The main difference between fibers and real threads is on the scheduling side: threads use *implicit*, *preemptive* scheduling while fibers use *explicit*, *non-preemptive* scheduling. This means that threaded code may be interrupted at any point, even in the middle of evaluating an expression, to give CPU cycles to code running in another thread. With fibers, these interruptions and context switches don't happen randomly; they are into the hands of the programmer who decides where his code is going to *yield* and give CPU cycles to other fibers.

The big advantage of fiber's explicit yielding is that the programmer does not need to protect critical code sections as long as they don't yield. Any piece of code that does not yield cannot be interrupted by other fibers. This means a lot less synchronization overhead.

But there is a flip side to the coin: threads are *fair*; fibers are not. If a fiber runs a long computation without yielding, it prevents other fibers from getting CPU cycles. A phenomenon known as *starvation*, and which is not new in node.js: it is inherent to node's event loop model; if a callback starts a long computation, it blocks the event loop and prevents other events from getting their chance to run.

Also, threads take advantage of multiple cores. If four threads compete for CPU on a quad-core processor, each thread gets 100% (or close) of a core. With fibers there is no real parallelism; at one point in time, there is only one fiber that runs on one of the cores and the other fibers only get a chance to run at the next yielding point.

Fibers – What for?

So, it looks like fibers don't bring much to the plate. They don't allow node modules to take advantage of multiple cores and they have the same starvation/fairness issues as the basic event loop. What's the deal then?

Fibers were introduced and are getting some love primarily because they solve one of node's big programming pain points: the so called *callback pyramid of doom*. The problem is best demonstrated by an example:

```
function archiveOrders(date, cb) {
  db.connect(function(err, conn) {
    if (err) return cb(err);
    conn.query("selectom orders where date < ?",
               [date], function(err, orders) {
      if (err) return cb(err);
      helper.each(orders, function(order, next) {
        conn.execute("insert into archivedOrders ...",
                     [order.id, ...], function(err) {
          if (err) return cb(err);
          conn.execute("delete from orders where id=?",
                       [order.id], function(err) {
            if (err) return cb(err);
            next();
          });
        });
```

```
      }, function() {
        console.log("orders been archived");
        cb();
      });
    });
  });
}
```

This is a very simple piece of business logic but we already see the pyramid forming. Also, the code is polluted by lots of *callback noise*. And things get worse as the business logic gets more complex, with more tests and loops.

Fibers, with Marcel's `futures` library, let you rewrite this code as:

```
var archiveOrders = (function(date) {
  var conn = db.connect().wait();
  conn.query("selectom orders where date < ?",
             [date]).wait().forEach(function(order) {
    conn.execute("insert into archivedOrders ...",
                 [order.id, ...]).wait();
    conn.execute("delete from orders where id=?",
                 [order.id]).wait();
  });
  console.log("orders been archived");
}).future();
```

The callback pyramid is gone; the signal to noise ratio is higher, asynchronous calls can be chained (for example `query(...).wait().forEach(...)`), etc. And things don't get worse when the business logic gets more complex. You just

write *normal* code with the usual control flow keywords (`if`, `while`, etc.) and built-in functions (`forEach`). You can even use classical `try/catch` exception handling and you get complete and meaningful stack traces.

Less code. Easier to read. Easier to modify. Easier to debug. Fibers clearly give the programmer a better comfort zone.

Fibers make this possible because they solve a tricky *topological* problem with callbacks. I'll try to explain this problem on a very simple example:

```
db.connect(function(err, conn) {
  if (err) return cb(err);
  // conn is available in this block
  doSomething(conn);
});
// Would be nice to be able to assign conn to a variable
// in this scope so that we could resume execution here
// rather than in the block above.
// But, unfortunately, this is impossible, at least if we
// stick to vanilla JS (without fibers).
```

The *topological* issue is that the `conn` value is only accessible in the callback scope. If we could transfer it to the outer scope, we could continue execution at the top level and avoid the pyramid of doom. Naively we would like to do the following:

```
var c;
db.connect(function(err, conn) {
```

```
  if (err) return cb(err);
  c = conn;
});
// conn is now in c (???)
doSomething(c);
```

But it does not work because the callback is invoked asynchronously. So `c` is still undefined when execution reaches `doSomething(c)`. The `c` variable gets assigned much later, when the asynchronous `connect` completes.

Fibers make this possible, though, because they provide a `yield` function that allows the code to wait for an answer from the callback. The code becomes:

```
var fiber = Fiber.current;
db.connect(function(err, conn) {
  if (err) return fiber.throwInto(err);
  fiber.run(conn);
});
// Next line will yield until fiber.throwInto
// or fiber.run are called
var c = Fiber.yield();
// If fiber.throwInto was called we don't reach this point
// because the previous line throws.
// So we only get here if fiber.run was called and then
// c receives the conn value.
doSomething(c);
// Problem solved!
```

Things are slightly more complex in real life because you also need to create a `Fiber` to make it work.

But the key point is that the `yield/run/throwInto` combination makes it possible to transfer the `conn` value from the inner scope to the outer scope, which was impossible before.

Here, I dived into the low level fiber primitives. I don't want this to be taken as an encouragement to write code with these primitives because this can be very error prone. On the other hand, Marcel's `futures` library provides the right level of abstraction and safety.

And, to be complete, it would be unfair to say that fibers solve just this problem. They also enable powerful programming abstractions like generators. But my sense is that the main reason why they get so much attention in node.js is because they provide a very elegant and efficient solution to the *pyramid of doom* problem.

Sponsored Ad

The *pyramid of doom* problem can be solved in a different way, by applying a CPS transformation to the code. This is what my own tool, `streamline.js`, does. It leads to code which is very similar to what you'd write with fiber's `futures` library:

```
function archiveOrders(date, _) {
  var conn = db.connect(_);
  flows.each(_, conn.query("selectom orders where date < ?",
                           [date], _), function(_, order) {
    conn.execute("insert into archivedOrders ...",
                 [order.id, ...], _);
    conn.execute("delete from orders where id=?",
                 [order.id], _);
  });
```

```
    console.log("orders been archived");
}
```

The signal to noise ratio is even slightly better as the `wait()` and `future()` calls have been eliminated.

And streamline gives you the choice between transforming the code into pure callback code, or into code that takes advantage of the `node-fibers` library. If you choose the second option, the transformation is much simpler and preserves line numbers. And the best part is that I did not even have to write the fibers transformation, Marcel offered it on a silver plate.

Wrapping up on fibers

In summary, fibers don't really change the execution model of node.js. Execution is still single-threaded and the scheduling of fibers is non-preemptive, just like the scheduling of events and callbacks in node's event loop. Fibers don't really bring much help with fairness/starvation issues caused by CPU intensive tasks either.

But, on the other hand, fibers solve the `callback pyramid of doom` problem and can provide a great relief to developers, especially those who have thick layers of logic to write.

Threads – What for?

As I said in the intro, threads landed into node this week, with Jorge's `thread_a_gogo` implementation (and I had a head start on them because Jorge asked me to help with beta testing and packaging). What do they bring to the plate? And this time we are talking about *real* threads, not the green kind. Shouldn't we be concerned that these threads will drag us into the classical threading issues that we had avoided so far?

Well, the answer is loud and clear: there is nothing to be worried about! These threads aren't disruptive in any way. They won't create havoc in what we have. But they will fill an important gap, as they will allow us to handle CPU intensive operations very cleanly and efficiently in node. In short, all we get here is bonus!

Sounds too good to be true! Why would these threads be so good when we had so many issues with threads before? The answer is simple: because we had the wrong culprit! The problems that we had were not due to the threads themselves, they were due to the fact that we had *SHARED MUTABLE STATE*!

When you are programming with threads in Java or .NET or other similar environments, any object which is directly or indirectly accessible from a global variable, or from a reference that you pass from one thread to another, is shared by several threads. If this object is immutable, there is no real problem because no thread can alter it. But if the object is mutable, you have to introduce synchronization to ensure that the object's state is changed and read in a disciplined way. If you don't, some thread may access the object in an inconsistent state because another thread was interrupted in the middle of a modification on the object. And then things usually get really bad: incorrect values, crashes because data structures are corrupted, etc.

If you have shared mutable state, you need synchronization. And synchronization is a difficult and risky art. If your locks are too coarse you get very low throughput because your threads spend most of their time waiting on locks. If they are too granular, you run the risk of missing some edge cases in your locking strategy or of letting deadlocks creep in. And, even if you get your synchronization right, you pay a price for it because locks are not free and don't scale well.

But *threads a gogo* (I'll call them *TAGG* from now on) don't share mutable state. Each thread runs in its own *isolate*, which means that it has its own copy of the Javascript code, its own global variables, its own heap and stack. Also, the API does not let you pass a reference to a mutable Javascript object from one thread to another.

You can only pass strings (which are immutable in Javascript) (*). So you are on the safe side, you don't run the risk of having one thread modify something that another thread is accessing at the same time. And you don't need synchronization, at least not the kind you needed around shared mutable objects.

(*) it would be nice to be able to share *frozen* objects across threads. This is not available in the first version of TAGG but this may become possible in the future. TAGG may also support passing buffers across thread boundaries at some point (note that this may introduce a limited, but acceptable, form of shared state).

I hope that I have reassured the skeptics at this point. As Jorge puts it, *these threads aren't evil*. And actually, they solve an important problem which was dramatized in a blog post a few months ago: node breaks on CPU intensive tasks. The blog post that I'm referring to was really trashy and derogative and it was making a huge fuss about a problem that most node applications won't have. But it cannot be dismissed completely: some applications need to make expensive computations, and, without threads, node does not handle this well, to say the least, because any long running computation blocks the event loop. This is where TAGG comes to the rescue.

If you have a function that uses a lot of CPU, TAGG lets you create a worker thread and load your function into it. The API is straightforwards:

```
var TAGG = require('threads_a_gogo');

// our CPU intensive function
function fibo(n) {
  return n > 1 ? fibo(n - 1) + fibo(n - 2) : 1;
}

// create a worker thread
var t = TAGG.create();
```

```
// load our function into the worker thread
t.eval(fibo);
```

Once you have loaded your function, you can call it. Here also the API is simple:

```
t.eval("fibo(30)", function(err, result) {
  console.log("fibo(30 result);
});
```

The function is executed in a separate thread, running in its own isolate. It runs in parallel with the main thread. So, if you have more than one core the computation will run at full speed in a spare core, without any impact on the main thread, which will continue to dispatch and process events at full speed.

When the function completes, its result is transferred to the main thread and dispatched to the callback of the `t.eval` call. So, from the main thread, the `fibo` computation behaves like an ordinary asynchronous operation: it is initiated by the `t.eval` call and the result comes back through a callback.

Often you'll have several requests that need expensive computations. So TAGG comes with a simple pool API that lets you allocate several threads and dispatch requests to the first available one. For example:

```
var pool = TAGG.createPool(16);
// load the function in all 16 threads
pool.all.eval(fibo);
// dispatch the request to one of the threads
pool.any.eval("fibo(30unction(err, result) {
```

```
  console.log("fibo(30 result);
});
```

TAGG also provides support for events. You can exchange events in both directions between the main thread and worker threads. And, as you probably guessed at this point, the API is naturally aligned on node's `Emitter` API. I won't give more details but the TAGG module contains several examples.

A slight word of caution though: this is a first release so TAGG may lack a few *usability* features. The one that comes first to mind is a module system to make it easy to load complex functions with code split in several source files. And there are still a lot of topics to explore, like passing frozen objects or buffers. But the implementation is very clean, very simple and performance is awesome.

Wrapping up on threads

Of course, I'm a bit biased because Jorge involved me in the TAGG project before the release. But I find TAGG really exciting. It removes one of node's main limitations, its inability to deal with intensive computations. And it does it with a very simple API which is completely aligned on node's fundamentals.

Actually, threads are not completely new to node and you could already write addons that delegate complex functions to threads, but you had to do it in C/C++. Now, you can do it in Javascript. A very different proposition for people like me who invested a lot on Javascript recently, and not much on C/C++.

The problem could also be solved by delegating long computations to child processes but this is costlier and slower.

From a more academic standpoint, TAGG brings a first bit of Erlang's concurrency model, based on *share nothing threads* and *message passing*, into node. An excellent move.

Putting it all together

I thought that I was going to write a short post, for once, but it looks like I got overboard, as usual. So I'll quickly recap by saying that fibers and threads are different beasts and play different roles in node.

Fibers introduce powerful programming abstractions like generators and fix the *callback pyramid* problem. They address a *usability* issue.

Threads, on the other hand, fix a hole in node's story, its inability to deal with CPU intensive operations, without having to dive into C/C++. They address a *performance* issue.

And the two blend well together (and — sponsored ad — they also blend with `streamline.js`), as this last example shows:

```
var pool = TAGG.createPool(16);
pool.all.eval(fibo);
console.log("fibo(30 pool.any.eval("fibo(30));
```

Kudos to Marcel and Jorge for making these amazing technologies available to the community.

Share this:

Share

Like

6 bloggers like this.

Related

Node.js: Awesome Runtime and New Age JavaScript Gospel
With 42 comments

Asynchronous Javascript - the tale of Harry
In "Asynchronous JavaScript"

TJ leaving node.js
In "node"

This entry was posted in Asynchronous JavaScript, Uncategorized. Bookmark the permalink.

55 Responses to *Fibers and Threads in node.js – what for?*

Alexis Coudeyras *says:*
March 11, 2012 at 23:21

Hi Bruno,

Great post.

Is there in node land a way to share data structures between workers/threads without cloning (either structured cloning or using a string to serialize the object) ?
I think chrome has added a way to share objects between workers without having to clone them, but when shared, the object was not accessible in the main or the others threads (to avoid synchronisation issue), i don't remember the name of this technic.

When you have big data structures and you want to split the work on them on multiple workers, cloning is an issue (because data is big) and "erasing" is an issue (because you can't share the data).

As you have said, if your data is immutable, you don't need to be share nothing, and since you have native popcycle immutability in javascript with object.freeze, i find it crazy to not use it.

There is something that is objectionable in the node.js (and erlang vm by the way) mindset. It seems to be like "you, developper, are stupid, so i won't allow you to do dangerous stuff". I prefer the mindset "you, developper, are stupid and lazy, so i will make an api that make the safe stuff easy to do and the dangerous stuff hard to do, but i will allow them, because maybe there is somebody who is smarter than me".
Shared mutable state between threads is evil because you need synchronisation and synchronisation is hard is just "almost" true.
You can share mutable state without synchronisation between threads if :

– you don't mutate it.
– each thread is responsible alone for a part of the state

If you want to split the work on a mutable array without cloning or freezing it for exemple, you can just create a wrapper taking the initial array, a start int and an end int, create a getAt(int idx) method (that add start to idx) and override the length method (return the end int instead of the real length of the array). And voilà, you've got a safe mutable shared data structure without cloning and synchronisation, you can create many instance of it, one for each worker. If your language support operator overloading, you can even do that transparently for the user. That's what scala does for it's parallel collection framework for exemple.
This is shared nothing, but it's user defined shared nothing, not imposed by technology shared nothing. And as Alex Payne put it (http://al3x.net/2010/07/27/node.html), when you really have to scale, you need every options on the table, not just the accepted best pratices. Of course, i will agree that most applications don't have big scalability issues, but some have, and more problematic : most of the time you discover those scalability issues late in the developpment process. If the only option you've got is "switch to C/C++", this can be a big problem if you don't have competences in C/C++.

More important, Node.js seems far away from other technologies about parallelism, either in the erlang vm or in .net (parallel framework) or in java (fork/join framework) you have high level abstraction that manage your worker pool, the decomposition of works into tasks, the assignment of tasks to workers with work stealing technics. In Java and in .net you can have synchronisation issues, but you shouldn't if you follow the guidelines (either by using a declarative model like in .net linq to objects queries, either by using a map/reduce model like in fork/join framework or be more functionnal like in scala). I don't see anything like that in node land. As you can see, even in non event looped technologies, we are far away from the old shared mutable state synchronisation way.

Even better, you can have very high level abstraction, like parallel collection in .net and scala, that manage the pool/task/steal works + make your algorithms (if they have no side effects) return the same results in mono thread mode and multi thread mode (think about the result of a parallel sort of an array for exemple), using transparently adapted parallel data structures. In Scala or in .net, i can parallelize list.sort(…).filter(…).map(…).groupBy(…) by adding .par before the list (or asParallel() in .net). Parallel collections don't solve all problems of course, but it's a cheap and (almost) safe way to handle a

lot of parallelism problems. Sometimes having a cheaper, faster and almost safe way to do stuff is better than having a more expensive, slower, safer way to do stuf.

The reactor pattern is a good fit for some applications, but not for all of them. It's the best fit for reverse proxies, applications with collaborations capabilities that have a lot of concurrent users, … But i don't think it's the best technology for classical business application or for cpu and memory intensive application without much io.
Node.js has the event loop model so engraved in his core, that i think node.js will soon encounter a big challenge. I think node.js strength is more in javascript + V8 + npm, than is the event loop model. And as more and more people will switch to node.js because of js/V8/npm, a lot of them will find that node.js is not a good fit for what they are doing and you'll have a lot of node.js forks. The multithreaded node.js for cpu intensive application, the big memory node.js for memory intensive application, the asynchronous as if it was synchronous node.js for business apps, …
I don't know how the node "core" community will react to that, if the really want to be the new php, they will have to be more open minded than they have been.

Reply

Chris Jacob *says:*
March 12, 2012 at 01:49

Excellent post. I've held off on diving into Node.js for a while now due to the "callback pyramid of doom" problem…. I knew it was only a matter of time until something more elegant came along.

Thanks for bringing me backup to speed with where Node.js is at, and for explaining the difference between Fibers and Threads so clearly!

I'm excited to take another look Node; so thank you for re-kindling my interest! ^_^

Reply

Bruno Jouhier *says:*

March 13, 2012 at 22:36

Hi Chris,

Both fibers and streamline.js have been around for more than a year. And there is at least one social web site in production that was built with streamline (and CoffeeScript). So these are solid bricks on which you can build real apps.

Reply

pyalot *says:*

March 12, 2012 at 12:46

I find the explicit notion of fibres backwards. There's a thing called "co-routines", and if you structure a simple API around them it looks like this:

```
var callable = function(a, b){
return a+b;
}

var c = new Coroutine(callable);
result = c.switch(1, 2) // is 3
```

Using co-routines you can do something rather convenient, you can build a scheduler, for instance on asynchronous I/O operations, and if you do, you can get completely synchronous, fibre unaware code like this:

```
var connection = DB.connect()
doSomething(connection)
```

Underneath a DB.connect would call some socket.connect/socket.read/socket.write, and underneath those the socket would be co-routine aware basically like this:
socket.read = function(data){
scheduler.switch(socket.READ);
write(self.fd, data);
}

I'm sure you see the logic in this, and I don't have to enumerate what the scheduler would do.

I'm rather strongly opposed to fibres/threads that can't be operated by code in a fashion that makes it invisible to application code.

Reply

Marcel Laverdet *says:*
March 14, 2012 at 02:48

You can do this in fibers.

Reply

sam *says:*
March 12, 2012 at 23:47

nice post! one question left.
you'v wrote about isolates but this feature is known to be canceled or to be more clearly isolates are known to be implemented later, much later.
so my question is now, if tagg relais on isolates how can tagg be over that threading feature or is it based on a unstable nodejs branch where isolates are implemented experimentally?

Reply

Bruno Jouhier *says:*

March 13, 2012 at 08:45

TAGG is just a user land module and it works with node.js 0.6 and 0.7. Isolates are a V8 feature. So anyone can write a C++ addon that takes advantage of them. There's no need to patch node.

Reply

sam *says:*

March 13, 2012 at 18:41

okay. i may just remembered something wrong about isolates. so tagg is seems to be the no pain in the ass solution for using multicore machines more efficiently 🙂 nice really nice!

Bruno Jouhier *says:*

March 13, 2012 at 22:25

Your memory is alright. There were talks about isolates support in node 0.7 but that got canceled.

From the little information I had, the design looked slightly different: each isolate was hosting a complete node runtime, while Jorge's threads are just hosting CPU intensive operations that don't do I/O.

shaun *says:*

March 15, 2012 at 04:36

You mentioned Marcel's Futures library working with Fibers.. can you let me know which library that is exactly? There are so many..

Reply

Bruno Jouhier *says:*

March 15, 2012 at 08:27

It is bundled with node-fibers (scroll down to FUTURES section). And, BTW, streamline also has futures: https://github.com/Sage/streamlinejs/wiki/Futures

Reply

shaun *says:*

March 21, 2012 at 08:55

Thank you.
And thanks for the blog post about the differences between the two libs.
Very helpful.

Hannes *says:*

March 25, 2012 at 20:02

Hi!
Thanks for the post! You clearly made it easier for me to work with threads and fibers in node.
One question though:
Is it possible to use other modules in a thread?
Because I can't get the thread to work if I'm using another module...
What I'm doing is:
x making a new thread and loading the code from a file
x in this file i want to poll a sensor over a serial-connection and send the message back to the main thread. Therefor I need the serial-module.

It all works fine standalone, but when I do a “require” in the thread-file it stops working, or at least I cannot send messages back to the main thread.

Any idea if it is possible to use other modules in threads?

Reply

Bruno Jouhier *says:*

March 25, 2012 at 20:42

Yes, `require` is not available in threads. I think that a module system would be a big plus but it hasn’t been implemented yet. So, today, you have to load code with a set of eval calls and you have to use closure tricks to encapsulate global vars. Not ideal but that’s how it is today…

Reply

Hannes *says:*

March 26, 2012 at 01:10

thanx for clearing this up!

Pingback: *Head, Tail and Callbacks in nodejs | The TapToLearn DevBlog*

Pingback: *Fibers and Threads in node.js – what for? « async I/O News*

Pingback: *Dealing concurrently with long running / blocking tasks in node.js | Nerdcode from the Ivory Tower*

Pingback: *Node.Js: Links, News And Resources (10) « Angel ”Java” Lopez on Blog*

Pingback: *Node.Js: Links, News And Resources (9) « Angel "Java" Lopez on Blog*

Menno van Lavieren *says:*

August 27, 2012 at 12:02

Hi,
The TAGG looks nice and takes in my opinion the right approach to threading, modeled on a 'web-service' style and then cut everything that isn't needed when running in process. First make something that is save and then adding features like modules to create a complete platform. I like the Nodejs/Erlang mindset where something that isn't save isn't allowed. In a big project there is always someone that will abuse an unsafe feature and will break everything. If it can't be proven correct it usually isn't.
But fibers on the other hand will add nothing useful to the table. The reintroduce the reentry problem that have plagued other event-loop based models. If right now in Nodejs I wrap a function in to a process.nextTick(), I will be sure it will be called only after the current call-stack has unwind. With Fibers someone else can call yield and cause the function to execute to early. I'll have to guard against this and maybe have to introduce a setTimeout to handle the case where the event-loop keeps executing my function in the current callstack.
The http library of Nodejs for example, now you will attach a event-handler to the onData event of a request inside the onRequest event-handler of your http server and be sure that you won't drop any data. But what if your onRequest handler calls a complex function that somewhere uses Fibers and yields the cpu? The onData event might trigger before the onRequest handler is done and has all the event handlers attached. With 'promises' or some new syntax in Coffee-script you are in control of yielding the cpu, with Fibers you are in a minefield.
With Fibers programming in Nodejs isn't simple anymore. It creates another trap in which every programmer will fall at some point.

Reply

Bruno Jouhier *says:*

August 27, 2012 at 13:04

I agree that you are in the danger zone if you use the low-level fiber primitives. But this is not the case if you either stick to the *futures* API, or if you use *streamline.js*.

If you stick to the *futures* API and discipline, the asynchronous functions that you write don't return values directly. Instead, they return a future and you have to call `wait()` on this future to get a value, and (very important) all functions that call `wait()` *must* return futures. So, the yielding points are explicit and marked by `wait()` calls at all levels. The only problem is that nothing enforces this discipline upon you and it is very easy to *not* return a future from a call that contains `wait()` calls. If you do this you enter the danger zone because you start hiding `yield` calls under the carpet.

With *streamline.js* you *cannot* hide the yielding points. It lets you call asynchronous functions the *sync* way but you have to pass the special _ marker. And you can only pass it from a function which has the _ marker in its parameter list. The streamline transformation engine will throw an exception if the marker is missing in the calling function's signature. So you have no way to hide the yielding points. They are explicit at all levels and this makes it very easy to *reason* about things like race conditions and re-entrancy problems (actually, easier than with callbacks). At least that's my experience with it.

So I think that streamline.js provides the right level of abstraction. It lets you use the normal language features (conditionals, loops, try/catch, chaining, composition, initializers, etc.) with asynchronous calls and yet it prevents you from hiding yielding points.

For me, fibers are a great runtime for streamline.js. And streamline is not tied to this runtime. It lets me choose between pure callbacks, fibers and generators (with different performance tradeoffs).

So the *new syntax in CoffeeScript* that you mention could very well be *streamline.js*. It works both with CoffeeScript and JavaScript.

Reply

Menno van Lavieren *says:*

August 27, 2012 at 15:17

It's a relief that the libraries on top of Fibers tackle this. I wouldn't like the nice and predictable world of Nodejs to get more complicated. It's a very subtle issue. And if things go wrong it is not easy to see how the problem really works or who is 'wrong'. I hope we won't end up in a situation where there is another coding guideline that says you should write all your code 'event-loop-reentry-proof'. Btw sorry my reply sounds a bit like a rant, I've to improve my writing skills.
I didn't have any specific syntax in mind, but streamline could very well be it.

Bruno Jouhier *says:*

August 27, 2012 at 16:01

Did not sound like a rant. The problem is real.

Tom Boutell (@boutell) *says:*

September 28, 2012 at 01:24

I'm late to this party, but the pyramid of doom problem has solutions that don't require hogging the CPU until everything is done or introducing preemptive threads.

The simplest:

```
function openDb()
{
db.connect(params, createTable);
}
```

```
function createTable(err)
{
db.createTable(params, insertFixtures);
}

function insertFixtures(err)
{
db.insertStuff(params, startServer);
}

function startServer(err)...
```

You get the idea.

This really isn't hard to read. The disadvantage is that you have to embed the name of the next function in each function.

You can use the async module to get around this in all sorts of great ways. async.waterfall is particularly relevant, so I'm going to paste its description. but it's not the only relevant option provided by async and similar modules.

waterfall(tasks, [callback])

Runs an array of functions in series, each passing their results to the next in the array. However, if any of the functions pass an error to the callback, the next function is not executed and the main callback is immediately called with the error.

Arguments

tasks – An array of functions to run, each function is passed a callback it must call on completion.
callback(err, [results]) – An optional callback to run once all the functions have completed. This will be passed the results of

the last task's callback.
Example

```
async.waterfall([
function(callback){
callback(null, 'one', 'two');
},
function(arg1, arg2, callback){
callback(null, 'three');
},
function(arg1, callback){
// arg1 now equals 'three'
callback(null, 'done');
}
], function (err, result) {
// result now equals 'done'
});
```

Reply

Bruno Jouhier *says:*
September 28, 2012 at 09:41

@boutell

These solutions are well known and I did experiment with them and others (promise libraries) two years ago, before writing streamline.js. My take at them is that they are ok if you're writing a tight I/O library or a small application. But if you have lots of business rules to write on top of asynchronous I/O layers they force you to write a lot of extra code and explode your logic. To get an idea, take the streamline tutorial

(https://github.com/Sage/streamlinejs/blob/master/tutorial/tutorial.md) and rewrite it with named functions or async. Then compare the results: readability, maintainability, elegance, robustness, etc.

Neither fibers nor streamline.js hog the CPU. The fibers library uses coroutines. Streamline.js generates callbacks (it writes them for you).

Preemptive threads address a different problem: CPU intensive computations. They have nothing to do with the pyramid of doom. That's one of the things I tried to explain in this post.

Reply

Blond Angel *says:*

October 10, 2012 at 22:42

heads up! Threads a go go is only for unix-based machines (linux, MacOS, etc). Currently not available for Windows (yet).

Reply

Mike Schinkel (@mikeschinkel) *says:*

October 27, 2012 at 22:52

Hi Bruno,

Thanks for the great article.

I'm relatively new to Node.js — most recently been developing in PHP for the past ~5 years — and am trying to figure out how to accomplish something and thing that streamlinejs might be the answer. Fibers standalone looks like it could be except fibers isn't supported on all Node.js hosts and that's a deal killer for me. Anyway I'm wondering if you could let me know if you think streamlinejs will address the issue I'm trying to resolve.

I'm trying to build an API proxy server in Node.js that execute a very simple Javascript-based DSL that basically makes it easy for low skilled programmers to call APIs and get back something meaningful. The GitHub project for this is at https://github.com/newclarity/concierge.

I want to allow people to submit scripts like the following and just have them work (this assumes a "friends" endpoint for an API):

```
var friends = $api.GET("friends",'12345is a user ID.
api.out(friends);
```

Or the script could be more elaborate and transform the data:

```
var friends = $api.GET("friends",'12345is a user ID.
var newFriends = [];
for(var i=0;i<friends.length;i++){
        newFriends[friends[i].ID] = friends[i].name;
}
api.out(newFriends);
```

Here's what $api might look like:

```
var $api = {
        'magic' : require('magic'),
        'GET' : function(resource,args) {
                var data;
                data = this.magic.get(self.applyArgs(self.resources['resource'],args));
```

```
                return data;
        },
        'applyArgs': function(template,args) {
                return 'logic to apply args here';
                },
        'resources' : {
                'friends' : 'http://api.example.com/user/{user_id}/friends',
                'events' : 'http://api.example.com/user/{user_id}/events'
        },
        out: function( out ) {
                this.magic.end(JSON.stringify(out));
        }
}
```

Which leads me to "magic" by which I mean I need something like that to implement the functionality I need in Node.js but in a non-blocking way. The absolute requirements for this are for the scripts to be exactly as I've presented them but the implementation of $api can be as complex as it needs to be to make this work.

Can I use streamlinejs and if yes would you mind giving me some hints as to how? Thanks in advance.

Reply

Bruno Jouhier *says:*

October 28, 2012 at 13:15

Hi Mike,

Fibers should be able to solve your problem completely (on its supported platforms).

Streamline.js can solve it but it requires a slight API change: you would need to add an _ parameter to all async calls. So the code that consumes your API would become:

```
var friends = $api.GET(_, "friends",'12345');  // 12345 is a user ID.
var newFriends = [];
for(var i=0;i<friends.length;i++){
        newFriends[friends[i].ID] = friends[i].name;
}
$api.out(_, newFriends);
```

and the code that implements it would look like:

```
var streams = require('streamline/lib/streams/server/streams');
var $api = {
        req: null,
        GET: function(_, resource,args) {
                this.req = streams.httpRequest(self.applyArgs(self.resources['resource'
                return this.req.end().response(_).checkStatus(200).readAll(_);
        },
        applyArgs: function(template,args) {
                return 'logic to apply args here';
        },
        resources: {
                'friends' : 'http://api.example.com/user/{user_id}/friends',
                'events' : 'http://api.example.com/user/{user_id}/events'
        },
        out: function(_, out) {
                this.req.write(_, JSON.stringify(out)).end();
```

```
        }
}
```

Note that the proposed API is problematic because it uses a global (`$api.req`) to pass context between `GET` and `out`). This could be fixed by rewriting it as:

```
function API() {
        var req;
        this.GET = function(_, resource, args) {
                req = streams.httpRequest(...);
                return req.end().response(_)...;
        }
        ...
        this.out = function(_, out) {
                req.write(_, JSON.stringify(out)).end();
        }
}
```

And you would consume it with `new API()` instead of `$api`.

Unfortunately, streamline needs the extra _ parameter because, unlike fibers, it does not use any C++ under the hood. So it cannot be quite as *magic*.

Also, note that people who would be using your API would be writing streamline code (*). So the `var friends = $api.GET(...) ...` code would need to be placed in a file with the `._js` extension.

(*) They could also use your API with plain JavaScript but then they would need to pass a real JavaScript callback (a `function(err, result) { ... }`) to all the async calls.

Reply

Bruno Jouhier *says:*

October 28, 2012 at 19:18

Oops. I replied too quickly. The code that I gave is wrong because the `req.write` call cannot be made on the same request. If the `GET` and `out` calls are independent, your API style is ok.

But the rest (use of _ parameter and the `GET` implementation) is ok.

Mike Schinkel (@mikeschinkel) *says:*

October 28, 2012 at 22:36

Hi Bruno,

Thanks for taking so much time to reply, I hadn't expected it but hope it helps others.
Unfortunately the requirement for our script is that it not expose and implementation specifics to the script writer/user in part because we want to flexibility to host on other platforms than Node.js if any relevant ones emerge and in part because we want the least amount of learning required for people to be able to write scripts. So I guess we'll have to continue to look for another solution.

But thank again for explaining so much in depth.

-Mike

Bruno Jouhier *says:*

October 29, 2012 at 19:44

Hi Mike,

If you want a sync style API your only choices are a) a coroutine extension to the JS runtime (fibers, ES6 yield) and b) a CPS transform (streamline.js, jscex, IcedCoffeeScript)

And unfortunately, you'll have to compromise somewhere. If you go with coroutines, you'll have limitations on supported platforms. If you go with CPS, you'll get portability (streamline.js runs browser side too) but you'll need a preprocessor. No miracle here.

Bruno

Reply

Mike Schinkel (@mikeschinkel) *says:*

October 29, 2012 at 20:27

Hi Bruno,

Thanks again for the detailed answer.

I may be stuck with the default CPS style and wait for a better coroutine solution from the general node community, if ever. Default CPS is better for this use case than the CPS transform approach although I can see the latter would be a very good approach for many other use-cases.

Thanks again.

-Mike

Reply

partridge *says:*

October 30, 2012 at 11:27

You know what, people didn't start using threads in the first instance, just for a laugh. They didn't stand around saying "hoho, what we want here is clearly a needless increase in complexity, because our jobs are not challenging enough".

If you don't need shared state, then you don't have a problem, and it's meaningless to speak of avoiding the pitfalls of threads. If you DO need shared state, then you are not going to avoid the pitfalls of threads if you want consistency.

Perhaps you can make the abstractions easier to understand, but you can't get shared state "for free".

It is no surprise to me that the node guys find themselves needing thread-like features, and it will be no surprise when they realise they need locking too.

Reply

Bruno Jouhier *says:*

October 30, 2012 at 22:50

Problem is not shared state, it is is shared *mutable* state.

Erlang, for example, has threads and shared *immutable* state, and message passing between threads (rather than locks).

I'm ready to bet that node will *never* have classical threads with shared mutable state and locks. This would break node (and JavaScript)'s fundamentals. But TAGG may get some traction as it is better aligned on node's model.

Reply

bunkertor *says:*
January 26, 2013 at 04:01

Reblogged this on Agile Mobile Developer.

Reply

Pingback: *How to run code using Node.js Fibers | BlogoSfera*

snoopyxdy *says:*
May 25, 2013 at 16:10

Hi Bruno,very nice post!
The tagg module have not updated for one year.Recently I have fork that,and modify it to support windows, linux and mac.So I named it tagg2. tagg2 also using node-gyp and it support nodejs 0.8.x and 0.10.x.

When I develop the tagg2, I find I can't use nodejs api in the thread,and the object in main thread could not share to the thread.Because one v8 isolate only can let one thread to visit it, so I can't using the thread to access the main thread's js object.

In order to solve the problem,I add the child process in to tagg2 module,it using the same api, and also add the process pool in tagg2 module.

I feeling multi threads in nodejs are not useful because of the v8.Using the multi threads in libuv and write the thread worker code in C++ are more wiser.

Welcome to have a try with tagg2.
npm install tagg2

github url
https://github.com/DoubleSpout/node-threads-a-gogo2

Reply

Pingback: *nodejs多线程，真正的非阻塞 » Romanysoft*

jeskeca *says:*
July 16, 2013 at 06:40

Be careful of using this “serializing” model for either futures or callback pyramids.

Async I/O’s primary benefit in high performance general purpose web-serving is *not* to run multiple user-request per CPU-thread (see below), but to remove serialization in multiple backend IO calls without introducing tons of threads (and the stack-memory they would consume). You want to use callbacks or futures to issue all your non-dependent IO calls simultaneously, and only once they have all been issued do you wait for any of them to respond. The only time “pyramids” should exist is when many calls are serially dependent on each-other, and this should be kept to a minimum because it increases user-latency.

For general purpose web-serving code, running many simultaneous requests cooperative/async on a single process/thread is not a good idea. Any CPU work in an async thread will *stall-all-other-tasks*.. which means if the code iterates over a big pile of database data for one user’s response, all other tasks on that thread/process will stall. If the server is something single purpose and highly tuned (like Jabber or SMTP) this won’t happen, but for general purpose web-code, don’t share processes with more than one user-request.

The preferred web scalability model is to run one only one user-request worker per CPU-thread, and use a separate dedicated async I/O spooler process to deal with buffering the IO and freeing up those workers as fast as possible (independent of client-download-speed). How many of these CPU-thread workers are needed depends on the codebase, but it is most often an integer multiple of CPU cores .. like cpu_cores * {2-3}. Then, within each worker, care should be taken to use async I/O to issue as many *parallel* backend calls as possible, to make their latencies overlap instead of additive.

Reply

Pingback: *nodejs多线程，真正的非阻塞 | Web开源笔记-专注Web开发技术,分享Web开发技术与资源*

Borna Novak *says:*

January 12, 2014 at 14:11

Seems like node.js community is going through the same cycle as did the Ruby community a couple years back after EventMachine was made (a reactor pattern high yield IO framework strikingly similar to node https://github.com/eventmachine/eventmachine).

EventMachine too implemented a thread pool for cpu intensive tasks (and as a workaround for handling blocking IO libs) which interacts with the event loop thread in a controlled way (through a callback running in the reactor thread),

and fibers came into play to solve the pyramid of doom in much the same way (https://github.com/igrigorik/em-synchrony).

Continuation of the story however,

is that dependence on a very limited eco-system and the whole reactor pattern model not playing nicely with VMs depending on threading to achieve parallelization (as opposed to C-based frameworks that depend on forking) led to a decline in use of that pattern.

Google's backing of node.js and browser's dependence on JavaScript to perform client-side processing will keep node.js up and running far beyond what it otherwise would in light of these problems, but the basic faults of node.js are strikingly similar to the faults of EventMachine that caused its slow decline out of favor – it would do well and good for the node community to raise awareness of these faults and start thinking on how to overcome them.

Ruby community answer for these issues was an actor-based framework heavily influenced by Erlang that utilizes threads to handle actor pools and fibers to prevent deadlocks – https://github.com/celluloid/celluloid.

The logical continuation of this pattern is a distributed actor system which uses 0MQ to implement the actor mailboxes: https://github.com/celluloid/dcell

And a curious twist came in the form of https://github.com/celluloid/celluloid-io which combines the reactor pattern with the actor pattern by allowing you to run reactors within actors

Reply

Bruno Jouhier *says:*

January 14, 2014 at 01:50

Celluloid looks cool. I wish node.js will get something similar in the near future. The closer we can get to the Erlang model, the better!

Thanks for the pointers.

Reply

fred *says:*

February 17, 2014 at 04:46

Based on my practice Node Fibers and packages built on top of it doesn't help to encapsulate an async function into a sync one and therefore cannot eliminate callback pyramid of doom, at least in my circumstance. I have elaborated my point in http://stackoverflow.com/questions/21819858.

Reply

Bruno Jouhier *says:*

February 17, 2014 at 08:37

Fred, I've posted an answer on stackoverflow.

Reply

fred *says:*

February 18, 2014 at 18:56

Thanks Bruno. I tried your answer and didn't work. The callback pyramid of doom you described is visible through code indent. But there is another type of callback pyramid of doom formed in the call stack. I added a scenario in my question to describe it in more details.

Bruno Jouhier *says:*

February 18, 2014 at 23:39

That's because the Express API is not a regular *continuation callback* API. More in my stackoverflow reply.

Reply

Jugz *says:*

March 7, 2014 at 14:23

Great Post! I was facing the heat from Pyramid of doom with Node.js (middleware) + express and I thought had to live this problem for ever. Fibers works perfectly.
hadnt heard about Isolated threading. Definitely a nice concept. Will match more in this space.

Reply

Jacques Clementi *says:*
April 16, 2014 at 15:23

Very useful! I've tried this to sort out a blocking I/O issue I have on node, but it didn't work as expected. Does anyone know if the blocking operation can be something like: instance.get(")
where instance is a native CPP V8 node extension?

Reply

HyperCaine *says:*
July 21, 2014 at 03:48

There's no need to use Fibres today to solve callback hell since Co-routines and Generators more than suffice – https://medium.com/code-adventures/callbacks-vs-coroutines-174f1fe66127, http://strongloop.com/strongblog/node-js-callback-hell-promises-generators/

Reply

Pingback: *The (Odd) State of Node.js and its Frameworks - Andrew Munsell's Blog*

Pingback: *The (Odd) State of Node.js and its Frameworks - Andrew Munsell*

Andy *says:*

October 28, 2016 at 17:20

Now that ES7 async/await is here, you should update this post. It seems to me that Fibers are mostly inferior to async/await and will eventually become a thing of the past given their disadvantages: use of native code and tendency to lag behind the latest version of node because of v8 API changes.

Reply

Bruno Jouhier *says:*

October 29, 2016 at 00:15

Node 7.0 was released 3 days ago and fibers run fine on it.

ES7 async/await is definitely an important step forwards but may not be the end of the story. Some day, JS may have to close the gap with go-lang and provide real coroutines.

Fibers are technically superior because they provide true deep continuations. Async/await restrict you to single-frame continuations and force you to have async/await keywords in every async function. Fibers let you write cleaner code: same general structure as ES7 async/await but a lot less keyword noise.

Reply

Pingback: *traiteur rabat*